"Express Mail" mailing label number:

EV 335 379 397 US

OBSTRUCTION-FREE MECHANISM FOR ATOMIC UPDATE OF MULTIPLE NON-CONTIGUOUS LOCATIONS IN SHARED MEMORY

Mark S. Moir,
Victor Luchangco and
Maurice Herlihy

CROSS-REFERENCE TO RELATED APPLICATION(S)

[1001] This application claims priority, under 35 U.S.C. § 119(e), of U.S. Provisional Application No. 60/396,152, filed 16 July 2002, naming Mark Moir, Victor Luchangco and Maurice Herlihy as inventors.

BACKGROUND

Field of the Invention

[1002] The present invention relates generally to coordination amongst execution sequences in a multiprocessor computer, and more particularly, to structures and techniques for facilitating non blocking implementations of shared data structures.

Description of the Related Art

[1003] Significant research effort has been applied in recent years to the development nonblocking implementations of shared data structures. Typically, this work is aimed at avoiding the numerous problems associated with the use of mutually exclusive locks when accessing shared data structures. These problems include deadlock, convoying and priority inversion and are generally well known in the art.

[1004] By using locks, operations on a shared data structure can prevent concurrent operations from accessing (parts of) the data structure for certain periods of time. In contrast, an operation in a nonblocking implementation of a shared data structure can be interrupted at any moment by another operation, and therefore the implementation must typically keep data consistent in *every* state, rather than simply ensuring that the data are consistent before releasing a lock. This presents a challenge, because if we want to change multiple parts of the data structure then we must prevent another operation from "seeing" some parts of the updates but not others.

[1005] Because current hardware architectures do not typically support an atomic modification of multiple, non-contiguous memory locations, it would be desirable to provide the illusion of this atomicity in software. Unfortunately, achieving this goal has proven difficult, particularly when we consider that nonoverlapping sets of atomic updates should not interfere with each other's performance if there is to be any hope of scalability of applications and data structure implementations that would employ the atomic updates.

SUMMARY

[1006] We present a novel technique for implementing obstruction-free atomic multi-target transactions that target special "transactionable" locations in shared memory. The programming interface for using operations based on these transactions can be structured in several ways, including as *n*-word compare-and-swap (NCAS) operations or as atomic sequences of single-word loads and stores (e.g., as transactional memory).

BRIEF DESCRIPTION OF THE DRAWINGS

[1007] The present invention may be better understood, and its numerous objects, features, and advantages made apparent to those skilled in the art by referencing the accompanying drawings.

[1008] **FIG. 1** depicts relationships between encodings of transactionable locations and transaction descriptors in accordance with some embodiments of the present invention.

[1009] **FIG. 2** is a flow diagram that highlights major flows in execution of a multitarget compare and swap (NCAS) operation in accordance with some embodiments of the present invention.

[1010] **FIG. 3** is a flow diagram that highlights major flows in execution of an illustrative ownership wresting sequence employed by a multi-target compare and swap (NCAS) operation in an attempt to acquire ownership of a target location thereof.

[1011] The use of the same reference symbols in different drawings indicates similar or identical items.

DESCRIPTION OF THE PREFERRED EMBODIMENT(S)

[1012] We have developed a new software-based technique for implementing multi-target atomic operations on an arbitrary number of non-contiguous memory locations using synchronization facilities available on many standard architectures. Such operations can be presented to the programmer in a variety of application programming interfaces (APIs), or simply employed (e.g., ad hoc) without definition of a standardized interface. Since the APIs tend to provide a useful descriptive context for illustrating our techniques, the description that follows assumes an API, though without limitation. One alternative API takes the form of two atomic operations: an atomic n-target compare-and-swap (NCAS) operation and an atomic load operation. The NCAS operation takes a sequence of n addresses, $a_1, \ldots a_n$, a sequence of n old values, $x_1, \ldots x_n$, and a sequence of n new values, $y_1, \ldots y_n$. If the operation returns *true*, then for each i, $1 \leq n$, the value at location a_i is equal to x_i, and the value at a_i is set to y_i. If the operation returns *false*, then no memory location is changed, and the operation returns *false*. We say a transaction (or NCAS) *succeeds* if it returns *true*, and *fails* if it returns *false*. The load operation simply takes an address and returns the value at that address.

[1013] An alternative API is a *transaction*: a sequence of single-target loads and stores that appear to take effect atomically with respect to other transactions. The programmer begins a transaction, executes a sequence of single-target loads and stores on behalf of that transaction, and then attempts to commit the transaction. If the commit succeeds, then none of that transaction's loads or stores appear to be interleaved with the loads and stores of other transactions. If the commit fails, then none of that transaction's stores will be visible to other transactions.

[1014] The proposed implementation is *obstruction-free*, meaning that if a thread *t* executes a transaction (or NCAS) and, at some point, *t* runs without interruption for long enough, then that transaction (or NCAS) will complete. The load operation is *wait-free*: it will return a value as long as the calling thread runs long enough (with or without interruption). The proposed implementation is also *linearizable*, implying that each transaction (or NCAS) appears to take effect instantaneously at some point between its invocation and response. For clarity of description, we focus on the NCAS implementation. Transforming this implementation to the transactional memory API is straightforward, and is discussed below.

Overview

[1015] We now describe our basic technique(s) in the context of a multitarget compare and swap (NCAS) operation. Each transactionable location may be "acquired" by an NCAS operation that wants to access it. In this case, we say that the NCAS operation *owns* the location. At most one NCAS operation may own a location at any time. Thus, concurrent NCAS operations that want to access the same location must compete for ownership of that location. In addition to a value, a transactionable location contains a pointer (or other suitable reference) to its owner, if any.

[1016] In an illustrative embodiment, each NCAS operation maintains a data structure called a *transaction descriptor*.

```
typedef  struct trans_s {
   status_t status;
   int size;                // size > 0
   value_t[] newvals;       // should have size elements
} trans_t;
```

[1017] The `size` field indicates the number of memory locations affected by the NCAS, and the `newvals` field is an array of `size` values that will replace the current values if the NCAS succeeds. The `status` field of a transaction descriptor indicates whether the current NCAS operation, if any, is on track to succeed.

```
typedef enum  {ACTIVE, SUCCESS, FAIL, LOST} status_t;
```

[1018] Transactionable locations can be represented as follows:

```
typedef  struct loc_s {   // Should be atomically CAS-able
```

```
      value_t  val;
      bool  held;
      trans_t  *trans;
      int  argno;
   } loc_t;
```

[1019] The `val` field encodes the value of the transactionable location (or in some exploitations, a reference to a location that encodes the value of the transactionable location). The `held` field encodes a Boolean status of the transactionable location. For example, a transactionable location is held if owned by an NCAS operation. The pointer `trans` identifies the NCAS operation (by transaction descriptor), if any, that owns the transactionable location. Finally, the field `argno` identifies the particular argument, $0 \leq \texttt{argno} < N$, where N is the size of the owning NCAS operation.

[1020] **FIG. 1** illustrates relationships between encodings of transaction descriptors and various transactionable locations in accordance with some embodiments of the present inventions. In particular, two active transactions corresponding to transaction descriptors **121** and **122** are illustrated. The first active transaction (*see* transaction descriptor **121)** corresponds to an N-way CAS, while the second active transaction (*see* transaction descriptor **122)** corresponds to a 2-way CAS (or DCAS). The first transaction targets N transactionable locations **110** and owns at least one of the transactionable locations, namely transactionable location **111**. One of the transactionable locations targeted by the first transaction, namely transactionable location **112**, is owned by the second transaction, which corresponds to transaction descriptor **122**. Transactionable location **113** is unowned.

[1021] To successfully complete, the first active transaction will need to acquire ownership of transaction location **112** in addition to each of the other transactionable locations it targets. Coordination amongst competing transactions will be understood in light of the description that follows.

[1022] **FIG. 2** illustrates major flows in execution of a multitarget compare and swap (NCAS) operation in accordance with some embodiments of the present invention. Initially, the status of a NCAS operation is ACTIVE. An NCAS operation first attempts (loop **201**) to acquire all the locations it wishes to update. If it determines that the current value of any of them is not equal to the expected value passed to the NCAS operation, then it fails **(202)** and returns *false*. Otherwise, once the NCAS operation has acquired all the specified locations, it attempts to change **(203)** its status from `ACTIVE` to `SUCCESS`. To properly synchronize with a concurrent operation that may be seeking to wrest ownership of a targeted location, the status update employs an appropriate synchronization primitive, e.g., a compare and swap (CAS) operation, a load-linked, store-conditional operation pair, etc. If the transaction succeeds in changing its status, then it writes back the new values and releases **(204)** the transactionable locations.

[1023] The instant at which a transaction's status field is changed to `SUCCESS` is the serialization point of a successful NCAS operation: the abstract values of all locations owned by that operation are changed atomically from the value stored in the locations to the new values stored in the transaction descriptor of the operation. The abstract value of a location changes only at the serialization point of a successful NCAS operation that targets the location.

[1024] An NCAS operation c_1 attempting to acquire a location that is owned by another NCAS operation c_2 must "wrest" ownership from c_2 in order to continue. **FIG. 3** illustrates major flows in execution of an illustrative ownership wresting sequence employed by a multitarget compare and swap (NCAS) operation in an attempt to acquire ownership of a target location thereof. Operation c_1 wrests ownership using an appropriate synchronization primitive, e.g., a compare and swap (CAS) operation, a load-linked, store-conditional operation pair, etc., **(301)** to change the status field of c_2 from *ACTIVE* to *LOST*. When c_2 attempts to complete the operation, it will fail, and must retry.

[1025] Assuming the synchronization primitive was successful (or that NCAS operation c_2 failed for some other reason), we attempt to update the transactionable location to reflect an unowned status, i.e., `held`=*false*. If the synchronization primitive was unsuccessful because the owning transaction was able to complete (updating it's status to *SUCCESS*), a new value for the transactionable location, which is stored in the owning transaction's descriptor, is used to update the value of the transactionable location. In either case, we attempt to update the transactionable location to reflect an unowned status, i.e., `held`=*false*. A synchronization primitive, e.g., a compare and swap (CAS) operation, a load-linked, store-conditional operation pair, etc., **(302)** is employed to mediate the attempt. If successful, the transactionable location value is returned **(303)**. If not, we retry **(304)**.

Exemplary NCAS Implementation

[1026] While the preceding description highlights certain aspects of an exemplary implementation of an NCAS operation, certain flows, conditions, behaviors, etc. are omitted for clarity of description. Accordingly, some of the omitted aspects will be better understood by persons of ordinary skill in the art based on review of the pseudocode that follows. In particular, certain details of ownership acquisition may be better understood with reference to the exemplary code.

[1027] The following code is merely illustrative, and based on the description herein persons of ordinary skill in the art will appreciate a wide range of suitable variations. Turning first to an NCAS operation:

```
bool NCAS(int n, (loc_t *)[] args, value_t[] evs,
          value_t[] nvs) {
// Assumes n>0, and args[0..n-1], evs[0..n-1] and
// nvs[0..n-1] are defined.  *args[0]..*args[n-1] are
// Tlocs to be accessed (read and modified).
// evs[i] is the value we "expect" to find in args[i]->val
// nvs[i] is the value written into args[i]->val if the
// NCAS succeeds,  args, evs and nvs are assumed to be
// local (not modified by other threads).
// They are not modified by this operation.
// Also assumes that args[i] != args[j] if i != j.
while (true) {
   /* Set up transaction descriptor */
   trans_t *t = new trans_t(n);  // transaction descriptor
                                 // for this NCAS
   t->status = ACTIVE;  // changes only once (to SUCCESS or
                        // LOST)
   t->size = n;  // size and newvals never change
                 // (after init)
   for (int i = 0; i < n; i++)
      t->newvals[i] = nvs[i];  // Can avoid copying (and
                               // allocating space for
                               // newvals) if nvs will not
                               // be recycled prematurely.
   if (!AcquireAll(n, args, evs, t)) {
      t->status = FAIL;
      return false;
   }
   CAS(&t->status, ACTIVE, SUCCESS);
```

```
      for (int  i = 0; i < n; i++)
         Release(args[i], t);
      if (t->status == SUCCESS)
         return true;
      else                          // t->status == LOST
         t->status = ACTIVE;        // try again
   } // end while (true)
   }
```

[1028] The locations, `*args[0]..*args[n-1]`, are transactionable locations to be accessed (read and modified) by the NCAS operation. A value `evs[i]` is the value we "expect" to find in the value field of a corresponding transactionable location, i.e., in `args[i]->val`, and `nvs[i]` is the value written into `args[i]->val` if the NCAS operation succeeds. Storage for the parameters `args`, `evs` and `nvs` is assumed to be local (i.e., not modified by other threads).

[1029] The NCAS operation employs a sequence of operations to acquire ownership of all targeted transactionable locations, while also ascertaining that value of each such targeted transactionable locations is as expected. By commingling the ownership acquisition and expected value checks, the illustrated implementation attempts to avoid unnecessary work. Pseudocode for three nested procedures `AcquireAll`, `Acquire` and `Wrest` employed by the NCAS operation follow:

```
bool  AcquireAll(int n, (loc_t *)[] args, value_t[] evs,
                 trans_t *t) {
// Assumes n > 0, and args[0..n-1] and evs[0..n-1] are
// defined.  *args[0]..*args[n-1] are Tlocs to be accessed
// (read and modified).  evs[i] is the (abstract) value we
// "expect" to find in *args[i].  args and evs are assumed
// to be local and are not modified.
// If AcquireAll returns true, then each *args[i] was
// <evs[i],true,t,i> at some time during the execution of
// this procedure.  (May be different times for different
// i's.)  If AcquireAll returns false, then for some i, the
// abstract value of *args[i] is not evs[i] at some time
// during the execution of this procedure.
// May "wrest" ownership of *args[i], aborting active NCAS
// operations.
for  (int  i = 0; i < n; i++) {
   if (  !Acquire(args[i],evs[i],t,i) ) {
      for ((int  j = 0; j <= i; j++)
         Release(arg[j], t);  // (Optional--helps other
                              // threads)
      return false;
   }
}
```

```
    return true;
} // end AcquireAll
```

[1030] As before, locations `*args[0]..*args[n-1]` are transactionable locations to be accessed (read and modified) by the NCAS operation. Value `evs[i]` is the (abstract) value we "expect" to find in `*args[i]`. Storage for `args` and `evs` are assumed to be local and are not modified.

[1031] If `AcquireAll` returns *true*, then each `*args[i]` had a value `<evs[i],true,t,i>` at some time during the execution of the procedure. That is, each targeted transactionable location encoded the expected value and was held (i.e., owned) by the current NCAS operation (transaction=`t`). Of course, the times may be different for different `i`'s. If `AcquireAll` returns false, then for some `i`, the abstract value of `*args[i]` is not `evs[i]` at some time during the execution of this procedure. `AcquireAll` invokes `Acquire` to check expected value and, if appropriate, acquire ownership for an individual transactionable location. Exemplary code for the `Acquire` procedure follows:

```
bool Acquire((loc_t *) arg, value_t ev, trans_t *t, int i) {
// If Acquire returns true, *arg was <ev,true,t,i>
// at some point during the execution of this procedure.
// If Acquire returns false, arg->val != ev at some point
// during the execution of this procedure.
// May "wrest" ownership of *arg, aborting active NCAS
// operations.
loc_t old;
while ((old = *arg) != <ev,true,t,i> )  { // not already
                                            // acquired
    if (old.held)
        old = Wrest(arg);
    if (old.val != ev)
        return false;
    if CAS(arg, old, <ev,true,t,i>) break;
}
return true;
} // end Acquire
```

[1032] If a transactionable location to be acquired by the NCAS operation is currently held by another transaction, the `Acquire` procedure invokes `Wrest` to make the location unowned, aborting (if necessary) an active competing transaction. The following code is exemplary:

```
loc_t Wrest (loc_t *arg)  {
// Makes the location unowned, if it isn't already.
// Does not change the abstract value of the location.
// Returns the new contents of *arg.
loc_t old, newv;
while ((old = *arg).held )  {
   if (old.trans->status == ACTIVE) {
      // Opportunity for backoff, in which case we should
      // reread status
      CAS(&old.trans->status, ACTIVE, LOST);
   }
   if (old.trans->status == SUCCESS)
      newv = <old.trans->newvals[old.argno],false,NULL,0>;
   else // old.trans->status == LOST or FAIL
      newv = <old.val,false,NULL,0>;
   if CAS(arg, old, newv)
      return newv;
}
return old;
} // end Wrest
```

[1033] A `Release` operation is also employed by the NCAS implementation to perform the appropriate value update and ownership clearing on successful completion of an NCAS operation. In addition, the release facility may be optionally employed even by a failing NCAS operation to clear ownership of acquired transactionable locations. Exemplary code for a `Release` operation follow:

```
void Release((loc_t *) arg, trans_t *t)  {
// Assumes t->status != ACTIVE
// Sets *arg to <--,false,NULL,0> if arg->trans == t
// (-- is abstract value)
if ((old = *arg).trans == t)  {
   if (t->status == SUCCESS)
      newv = <t->newvals[old.argno],false,NULL,0>;
   else
      newv = <old.val,false,NULL,0>;
   CAS(arg, old, newv)
}
} // end Release
```

[1034] Finally, while we have focused on description of an NCAS operation, persons of ordinary skill in the art will appreciate that applications will typically need to load values from transactionable locations as well. Because the value of a transactionable location may reside in the transactionable location or with an owning transaction, implementation in accordance with the following pseudocode may be employed to obtain the appropriate version.

```
value_t trans_Load( loc_t *l )   {
   loc_t tloc = *l;
   if (!tloc.held)
      return tloc.val;
   if (tloc.trans->status  !=  SUCCESS)
      return tloc.val;
   return tloc.trans->newvals[tloc.argno];
}
```

[1035] The `trans_Load` operation simply reads the location and, if the location is unowned, returns the value stored there. If the location is owned, then the operation reads the status of the owner. If the owner has successfully completed, then the load returns the value stored in the corresponding entry of the array of new values stored in the owner's transaction descriptor. Otherwise, the `trans_Load` returns the value it read in the location.

Correctness

[1036] To see that the operations are correctly implemented, we show that a `trans_Load` operation returns a value that was the abstract value of the location at some time during its execution (and does not change the abstract value of the location), that for a successful NCAS operation, there is a time during its execution that the abstract values of all the locations matched the expected values passed into the NCAS operation and were atomically updated to the new values passed into the operation, and that for a failed NCAS operation, some location specified by the operation did not have the expected value at some time during its execution.

[1037] To see this, recall that the abstract value of a location is determined by the contents of the location and by the contents of the descriptor of the owning transaction, if any. Specifically, the abstract value of an unowned transactionable location is the value stored in that location. The abstract value of an owned transactionable location depends on the status of the owner: If the owner has not succeeded, the abstract value of the location is still the value stored in that location. If the owner has succeeded, the abstract value is the value stored in the corresponding entry of the array of new values stored in the owner's transaction descriptor.

[1038] The key to the correctness argument is that when an NCAS operation changes its status from *ACTIVE* to *SUCCESS*—this point is the serialization point of a successful NCAS operation—it still owns all the locations it acquired. This property is guaranteed because an operation that owns a location only loses ownership (without releasing the location) when the location is wrested by another operation. In this case, the other operation must first change the first operation's status to *LOST*. Thus, the first operation's attempt to change its status from *ACTIVE* to *SUCCESS* will fail, and the operation must retry. It is also important to note that, once an operation has succeeded, its status will be *SUCCESS* thereafter.

[1039] A failed NCAS operation is serialized at the point that it reads (inside the `Acquire` procedure) a location whose value is not the expected value passed into the operation. The value stored in the location is the abstract value of the location because `Acquire` ensures that the location is unowned before checking its value against the expected value.

[1040] A load that finds a location unowned, or finds that the owner has not yet succeeded, is serialized at the point that it read the location. (The owner, if any, had not yet succeeded at this point, if it has not succeeded when the load checks its status.) If the location is owned by an NCAS operation that has succeeded at the time the load operation checks its status, then the load is serialized at the later of the time that it read the location and the time (immediately after) the owner changed its status to *SUCCESS*. This instant always occurs during the execution of the load operation (after it read the location and before it read the status of the owner), and the value returned is the abstract value at that time because, by the key correctness property above, an NCAS operation that changes its status to *SUCCESS* owns all the locations it acquired (including the location being loaded) at the time that it updates its status.

[1041] The load operation is clearly wait-free, as it has no loops. Informally, it is easy to see that the NCAS operation is obstruction-free because once an operation runs alone, eventually either it will find a location that does not have the expected value, in which case it will fail, or it will wrest all its desired locations from other operations that may own them, find that they all have the expected values, and succeed in changing its status to *SUCCESS*. Verifying this rigorously is a straightforward task: every while loop completes in at most one full iteration if it is executed in isolation.

[1042] Note that two threads attempting to NCAS the same location may indefinitely cause each other to retry, even if they both expect the correct value, which never changes during the execution of their NCAS operations. This can occur only if each wrests the location from the other before the other is able to successfully complete; for this to happen, both threads must take steps. As a practical matter, such conflicts can be avoided by standard means such as exponential back-off or queuing.

Memory Management

[1043] The only data structures allocated here are transaction descriptors: each transaction gets a new transaction descriptor. These transaction descriptors can be recycled using nonblocking memory management techniques such as described in commonly owned, co-pending U.S. Patent Application No. 10/340,156, filed January 10, 2003, naming Mark S. Moir, Victor Luchangco and Maurice Herlihy as inventors.

Extensions to Larger Transactionable Locations

[1044] For convenience, we have thus far presented our techniques assuming that a memory location sufficient to contain a `loc_t` record can be atomically accessed by load and CAS instructions. If this is not the case, several alternative implementations are possible consistent with the techniques we have described.

[1045] For example, we may employ an additional level of indirection so that, rather than using a CAS instruction to modify `loc_t` records, we instead use a CAS instruction to modify pointers to buffers that are large enough to contain `loc_t` records. It is safe to reuse one of these buffers only after it has been determined that no thread will subsequently read from the buffer before it is reused. Such determination can be made by standard garbage collectors, or by nonblocking memory management techniques such as those described in commonly owned, co-pending U.S. Patent Application No. 10/340,156, filed January 10, 2003 and naming Mark S. Moir, Victor Luchangco and Maurice Herlihy as inventors, the entirety of which in incorporated herein by reference. Given this arrangement, and because the buffers are not modified after they are initialized and before they are reused, we can consider the load of a pointer to a buffer as an atomic load of the contents of that buffer, and our implementation works as described previously.

[1046] Another option is to relax the requirement that a location contains the old value of the location while that location is owned by a transaction. This way, at any point in time, each location contains *either* its value (if the location is not currently owned) *or* a reference to the owning transaction descriptor. In this case, to facilitate the wresting of ownership by one transaction from another, a transaction that acquires ownership of the location first stores the value that will be displaced by that acquisition somewhere that it can be found by the wresting transaction; a natural place to store the displaced value is in the transaction descriptor. This approach assumes that there is some effective method for distinguishing application values from references to transaction descriptors. This can be achieved for example by "stealing a bit" from application pointers, using well-known techniques for aligned allocation so that the least significant bit of every pointer is zero. Another possibility exists if the application values are known to be pointers returned to the application by a memory allocator. In this case, because the memory allocator will not return to the application a pointer to a transaction descriptor already allocated by the NCAS or transactional memory implementation, we can distinguish application values from references to transaction descriptors by keeping track of the addresses of all transaction descriptors. This can be made efficient by allocating all transaction descriptors in a single block, so that distinguishing them from application values is merely an address range comparison.

[1047] In the case that we "displace" application values into transaction descriptors when acquiring ownership, it is more difficult to provide a wait-free load operation because we may have to repeatedly "chase" the displaced value from one transaction to another. However, it is still straightforward to provide an obstruction-free load operation because this chasing will occur only in the presence of contention with concurrent transactions.

Transactional Memory Variations

[1048] It is straightforward to convert the illustrated NCAS implementation to a transactional memory implementation. As before, each transaction must acquire a location before loading from that address or storing to it. Instead of keeping a fixed-size `newval` field in the transaction descriptor, the transaction can keep a table of new values indexed by address. When a location is acquired, the transaction copies the address and its current value into the table. To load from that address, the transaction returns the corresponding value from the table. To store to that address, the transaction modifies the corresponding value from the table. To commit a transaction, the transaction moves the value from the table to the memory location, and resets the `held` and `trans` field as before. One transaction wrests a value from another as before.

[1049] The principal difference between the transactional memory API and the NCAS API is that the number of locations affected by a transaction need not be declared in advance. Earlier work in the area of software transactional memory required that a transaction predeclare the memory locations it would update in order to ensure that some transaction always made progress. The obstruction-free property, however, does not make such strong guarantees, so the programmer is free to choose memory locations on the fly.

Contention Management Strategies

[1050] Despite our advocacy of obstruction-free synchronization, we do *not* expect progress to take care of itself. On the contrary, we have found that explicit measures are often necessary to avoid starvation. Obstruction-free synchronization encourages a clean distinction between the obstruction-free mechanisms that ensure correctness (such as conflict detection and recovery) and additional mechanisms that ensure progress (such as adaptive backoff or queuing).

[1051] In our multitarget transaction implementation, progress is the responsibility of a *contention manager*, which may be separate from, or modular with respect to the transaction implementation itself. Each thread has its own contention manager instance, which it consults to decide whether to force a conflicting thread to abort. In addition, contention managers of different threads may consult one another to compare priorities and other attributes.

[1052] The correctness requirement for contention managers is simple and quite weak. Informally, any active transaction that asks sufficiently many times must eventually get permission to abort a conflicting transaction. More precisely, every call to a contention manager method eventually returns (unless the invoking thread stops taking steps for some reason), and every transaction that repeatedly requests to abort another transaction is eventually granted permission to do so. This requirement is needed to preserve obstruction-freedom: A transaction *T* that is forever denied permission to abort a conflicting transaction will never commit even if it runs by itself. If the conflicting transaction is also continually requesting permission to abort *T,* and incorrectly being denied this permission, the situation is akin to deadlock. Conversely, if *T* is eventually allowed to abort any conflicting transaction, then *T* will eventually commit if it runs by itself for long enough.

[1053] The correctness requirement for contention managers does *not* guarantee progress in the presence of conflicts. Whether a particular contention manager should provide such a guarantee - and under what assumptions and system models it should do so-is a policy decision that may depend on applications, environments, and other factors. The problem of avoiding livelock is thus delegated to the contention manager. The `Wrest` operation, detailed above, illustrates one suitable opportunity for contention management by backoff or other contention management technique.

Other Embodiments

[1054] While the invention(s) is(are) described with reference to various embodiments, it will be understood that these embodiments are illustrative and that the scope of the invention(s) is not limited to them. Terms such as always, never, all, none, etc. are used herein to describe sets of consistent states presented by a given computational system, particularly in the context of correctness proofs or discussions. Of course, persons of ordinary skill in the art will recognize that certain transitory states may and do exist in physical implementations even if not presented by the computational system. Accordingly, such terms and invariants will be understood in the context of consistent states presented by a given computational system rather than as a requirement for precisely simultaneous effect of multiple state changes. This "hiding" of internal states is commonly referred to by calling the composite operation "atomic", and by allusion to a prohibition against any process seeing any of the internal states partially performed.

[1055] Many variations, modifications, additions, and improvements are possible. For example, while application to particular multitarget transactions and particular implementations thereof have been described in detail herein, applications to other transactions, multitarget operations and implementations will also be appreciated by persons of ordinary skill in the art. In addition, more complex shared object structures may be defined, which exploit the techniques described herein. Other synchronization primitives may be employed. Plural instances may be provided for components, operations or structures described herein as a single instance. Finally, boundaries between various components, operations and data stores are somewhat arbitrary, and particular operations are illustrated in the context of specific illustrative configurations. Other allocations of functionality are envisioned and may fall within the scope of the invention(s).

[1056] In general, structures and functionality presented as separate components in the exemplary configurations may be implemented as a combined structure or component. Similarly, structures and functionality presented as a single component may be implemented as separate components. These and other variations, modifications, additions, and improvements may fall within the scope of the invention(s).